Solutions to Quiz 1 (October 24, 2016)

Problem 1 (Specifications) (**18 points**).
Consider this partial specification:

```
/**
 * Shift the values in inputs, wrapping around the ends of the array.
 * For example, [2, 4, 6, 8, 10] shifted by 2 to the right is [8, 10, 2, 4, 6].
 *
 * @param inputs array to shift, ...
 * @param delta amount to shift by, ...
 * @return inputs shifted by ...
 */
public static int[] shiftArray(int[] inputs, int delta)
```

Here are two versions of this specification that fill in the missing pieces:
Specification **A**:

```
@param inputs array to shift, must be non-empty
@param delta amount to shift by, must be a positive integer
@return inputs shifted by delta units to the right
```

Specification **B**:

```
@param inputs array to shift, must be non-empty
@param delta amount to shift by
@return inputs shifted by delta units,
        to the right if positive, to the left if negative
```

For each question, answer by filling in the blanks. You may leave blanks empty if desired, and you may use the same answers more than once.

(a) Complete a specification that is weaker than **A**:

`@param inputs array to shift,` _____

`@param delta amount to shift by,` _____

`@return inputs shifted by` _____

_____

Solution. Strengthen A's precondition (e.g. `delta = 5`) or weaken the postcondition (e.g. shifted by at most `delta` units to the right). ■

(b) Complete a specification that is stronger than **A** and weaker than **B**:

`@param inputs array to shift,` _____

`@param delta amount to shift by,` _____

`@return inputs shifted by` _____

_____

Solution. Precondition between A and B (e.g. `delta` non-negative). ■

(c) Complete a specification that is stronger than **A** but neither stronger nor weaker than **B**:

`@param inputs array to shift,` _____

`@param delta amount to shift by,` _____

`@return inputs shifted by` _____

_____

Solution. Weaken the precondition and choose a different postcondition (e.g. any `delta`, but shift by `Math.abs(delta)` to the right). ■

Problem 2 (Testing) (**28 points**).
Given this specification:

```
/**
 * If text contains bound, split into two substrings a and b, where:
 * - a ends with bound, b starts with bound, and
 * - if a = a_begin + bound, and b = bound + b_end, then
 *     text = a_begin + bound + b_end.
 * Otherwise, don't split.
 *
 * For example: splitWithBound("1a2", 'a') is { "1a", "a2" }
 *              splitWithBound("123", 'a') is { "123" }
 *
 * @param text text to split
 * @param bound character to split on
 * @return a set containing some a and b as defined above, if any,
 *         or just text if none
 */
public static Set<String> splitWithBound(String text, char bound)
```

(a) For each test case below, **write YES or NO** in the first box to say whether the test is a valid test for `splitWithBound`. If the test is not valid, **explain why** or provide a corrected test.

1. `"zaz"` × `'a'` → `{ "az", "za" }`
   Valid? Reason if invalid:
2. `"az"` × `'a'` → `{ "a", "az" }`
   Valid? Reason if invalid:
3. `"zaaz"` × `'a'` → `{ "za", "aaz" }`
   Valid? Reason if invalid:
4. `"aa"` × `'a'` → `{ "a" }`
   Valid? Reason if invalid:
5. `"a"` × `'a'` → `{ "a" }`
   Valid? Reason if invalid:
6. `""` × `'a'` → `{ "a" }`
   Valid? Reason if invalid:

Solution. 1, 2, and 5 are valid.

For 3, output `{ "zaa", "az" }` is also possible.

For 4, the output would be `{ "a", "aa" }`.

For 6, the output would be `{ "" }`. ■

(b) Start implementing a systematic testing strategy for this function by writing **one good partitioning** of the input space on **input `text` alone**. Your partitions for this question should not mention `bound`. In addition, the subdomains in your partitioning **must not overlap**.

Solution. E.g. `text.length()` $= 0, = 1, > 1$. ■

(c) Now, write **one good partitioning** of the input space on **location(s) where `bound` appears in `text`**. In addition, the subdomains in your partitioning **must not overlap**.

Solution. E.g. `bound` appears at beginning only; beginning and middle; beginning and end; beginning, middle, and end; middle only; middle and end; end only; not at all. ■

(d) Suppose we instead make this function an instance method of an immutable class `Splitter`.

An instance of `Splitter` uses a fixed `bound`:

```
public class Splitter {
    // Make a Splitter that uses the given bound
    public Splitter(char bound) { ... }

    // Split with bound, as above
    public Set<String> splitWithBound(String text) { ... }
}
```

Which of the following are true? (choose all that apply)

A. Unit tests of `Splitter(char)` should not call `splitWithBound(String)`

B. Unit tests of `splitWithBound(String)` should not call `Splitter(char)`

C. This immutable class doesn't make sense, because `splitWithBound` can be called with different values of `text`

D. The `splitWithBound` operation of `Splitter` is an *observer* according to our classification

E. *none of the above*

Solution. D. ■

Problem 3 (Debugging) (**12 points**)**.**
You've written the following program:

```
1 public class Sentences {
2
3     public static void main(String[] args) {
4         String s = "no sentence. Sentence. also no sentence";
5         System.out.println(findSentence(s.toCharArray()));
6     }
```

```
 7
 8      // Return the text (without period) of the first sentence in chars.
 9      // A sentence begins with a capital letter, ends with a period, and
10      //   doesn't contain any other periods.
11      public static char[] findSentence(char[] chars) {
12          int start = 0;
13          boolean scanning = false;
14          for (int i = 0; i < chars.length; i++) {
15              if (Character.isUpperCase(chars[i]) && ! scanning) {
16                  start = i;
17                  scanning = true;
18              }
19              if (chars[i] == '.' && scanning) {
20                  return ArrayUtils.copySubarray(chars, start, i);
21              }
22          }
23          return new char[0];
24      }
25  }
```

You run it, and the output is:

```
Sentence. also no sen
```

But you expected the output to be just:

```
Sentence
```

It's broken!

Just then, your boss shows up at your desk: *We can't afford all these CPU cycles you're burning! Run your program one more time, then just fix it already!*

(a) Choose your own adventure!

Option 1	Option 2	Option 3
Run again with input: `21.Abc.`	Run again with input: `Abc.123`	Run again with input: `21.Abc.123`

Raise your hand and quietly inform the responding TA that you would like to explore **Option 1, 2, or 3**. The TA will highlight your chosen option and the corresponding output on the next page.

You may not change your choice or ask again.

```
21.Abc

1.Abc

Abc

Abc.

Abc.1

Abc.12
```

Only the output highlighted by the TA is produced by your chosen input.
You may ignore all the other possibilities.

```
Abc.123

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at Sentences.findSentence(Sentences.java:15)
    at Sentences.main(Sentences.java:5)

Exception in thread "main" java.lang.IndexOutOfBoundsException
    at ArrayUtils.copySubarray(ArrayUtils.java:18)
    at Sentences.findSentence(Sentences.java:20)
    at Sentences.main(Sentences.java:5)

Exception in thread "main" java.lang.IllegalArgumentException
    at ArrayUtils.copySubarray(ArrayUtils.java:15)
    at Sentences.findSentence(Sentences.java:20)
    at Sentences.main(Sentences.java:5)
```

Solution. The outputs are:

Option 1: `IndexOutOfBoundsException` from `copySubarray` (`ArrayUtils.java:18`)

Option 2: `Abc`

Option 3: `Abc.12` ■

(b) Given all the information you have, what is the bug? Write your single best hypothesis clearly and succinctly, giving evidence to support it.

Solution. `ArrayUtils.copySubarray` requires a start index and a length, but is called with a start index and an end index. ■

(c) Propose a fix. If you can, identify a line or lines of code, and say how they should be changed.

Solution. `return ArrayUtils.copySubarray(chars, start, i-start);` ■

Problem 4 (ADTs) (**28 points**).
Consider this abstraction function, rep invariant, and argument for safety from rep exposure for a class `Document` to represent immutable text documents with section headings, where each heading is a single line, followed by a single line of text:

AF
represents the document heading $\texttt{headings}_0$, then text $\texttt{contents}_0$, then heading $\texttt{headings}_1$, then text $\texttt{contents}_1$, ..., up to `length`-1; where $\texttt{headings}_i$ is the i^{th} element in `headings`, and similarly for `contents`

RI
`length` = `headings.size()` = `contents.size()`
none of the elements in `headings` or `contents` contain newlines

Safety from rep exposure
all fields are final
`length` is primitive
`headings` and `contents` are created in constructors, never returned directly to clients, and contain immutable objects

(a) What is the rep of `Document`? Use the most reasonable choices from the standard library, and write valid lines of Java.

Solution.

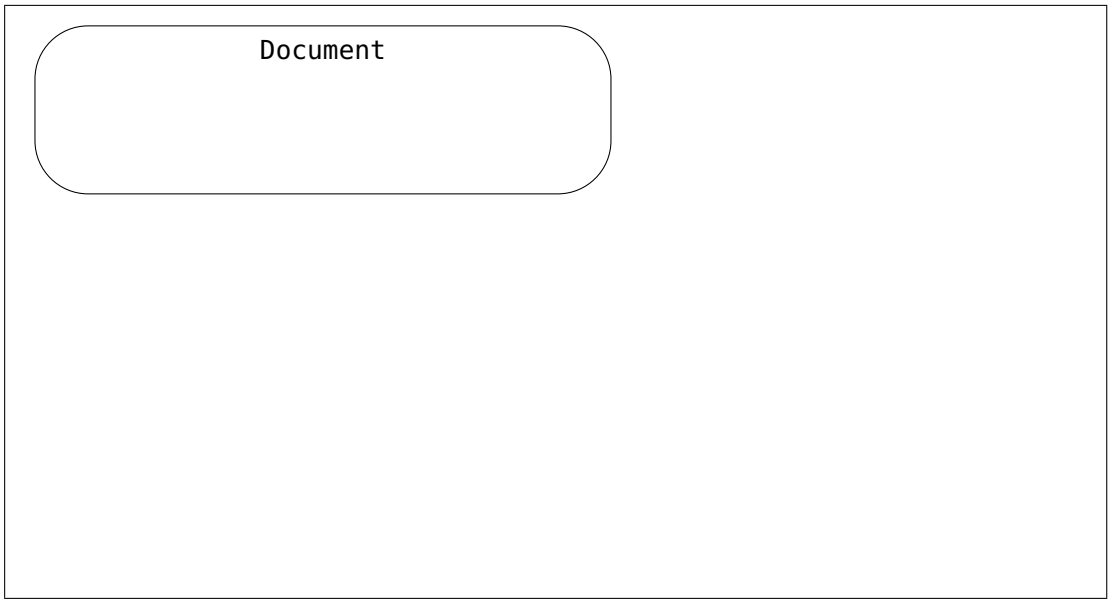
```
private final int length;
private final List<String> headings;
private final List<String> contents;
```

■

(b) Draw a snapshot diagram, including as much information as you can, for the concrete representation of the abstract value:

Hello
Greetings!

Goodbye
Farewell.



Solution. Immutable `Document`. Immutable field `length` has value `2`. Immutable fields `headings` with `List` value containing the two heading strings, and `contents` with `List` value containing the two content strings. ■

(c) If all arrays, collections, and strings in the rep (if any) are empty, all primitive values (if any) are their default value, and all other objects (if any) are **`null`**, what is the abstract value? If none, explain why.

Solution. The empty document, with no headings or text. ■

(d) Suppose we implement `Document` with a different representation. Here are the rep invariant and safety from rep exposure argument:

RI
`length*2` = `pieces.size()`
none of the elements in `pieces` contain newlines

Safety from rep exposure
all fields are final
`length` is primitive
`pieces` is created in constructors, never returned directly to clients, and contains immutable objects

Write an abstraction function that works with this rep and rep invariant:

AF

Solution. Represents the document: heading `pieces`$_0$, then text `pieces`$_1$, then heading `pieces`$_2$, then text `pieces`$_3$, ..., up to `length*2-1`; where `pieces`$_0$ is the i$^{\text{th}}$ element in `pieces`. ■

(e) Write a **recursive datatype definition** for `Document` that uses **two** concrete variants, one of which must be **`Empty`**, to represent the same abstract values as above using a recursive structure:

```
Document =

           +
```

Solution. `Document = Cons(heading:String, text:String, rest:Document) + Empty()` ■

Problem 5 (Multiple Choice) (**14 points**).

(a) Which of the following statements about equality are true? (choose all that apply)

A. If you override the `equals()` method for an immutable ADT, you should also override `hashCode()`

B. You should implement observational equality for all ADTs

C. Returning a constant integer is a valid implementation of `hashCode()` because the spec of `hashCode()` specifically allows magic number implementations.

D. Using the abstraction function AF, we can define equality as: a equals b if and only if $AF(a) = AF(b)$

E. Unless we override it, `equals()` implements reference equality

Solution. A and D.

B is false, only for immutable ADTs. For C, returning a constant is legal, but not because of a specific exception. ■

(b) Which of the following best describes the benefit of declaring a variable using an interface type, as in this example:

```
List<String> words = new ArrayList<>();
```

(choose **one** best answer)

A. SFB: the `List` interface isolates our code from bugs in `ArrayList`

B. SFB: this requires `ArrayList` methods to meet specifications in `List`

C. ETU: readers may not know what an `ArrayList` is, but they will recognize `List`

D. RFC: we can change the code to use, *e.g.*, `LinkedList` instead of `ArrayList`

Solution. D.

A and B are incorrect. C may be true, but is not as compelling as D. ■

(c) You are implementing a `Building` ADT. Buildings have a list of rooms, which you represent with:

```
private final List<Room> rooms;
```

`Room` is immutable. Which of the following implementations for a `getRooms()` method will avoid rep exposure? (choose all that apply)

```
public List<Room> getRooms() { return ???; }
```

A. `rooms`

B. `Collections.ummodifiableList(rooms)`

C. `new ArrayList<>(this.rooms)`

D. `Optional.of(this.rooms)`

E. `this.rooms`

Solution. B and C.

B uses an umodifiable wrapper. C makes a defensive copy. ■